binary manual page - Tcl Built-In Commands

tcl.tk/man/tcl/TclCmd/binary.htm

NAME

binary — Insert and extract fields from binary strings

SYNOPSIS

binary decode *format* ?*-option value ...*? *data*
binary encode *format* ?*-option value ...*? *data*
binary format *formatString* ?*arg arg ...*?
binary scan *string formatString* ?*varName varName ...*?

DESCRIPTION

This command provides facilities for manipulating binary data. The subcommand **binary format** creates a binary string from normal Tcl values. For example, given the values 16 and 22, on a 32-bit architecture, it might produce an 8-byte binary string consisting of two 4-byte integers, one for each of the numbers. The subcommand **binary scan**, does the opposite: it extracts data from a binary string and returns it as ordinary Tcl string values. The **binary encode** and **binary decode** subcommands convert binary data to or from string encodings such as base64 (used in MIME messages for example).
Note that other operations on binary data, such as taking a subsequence of it, getting its length, or reinterpreting it as a string in some encoding, are done by other Tcl commands (respectively **string range**, **string length** and **encoding convertfrom** in the example cases). A binary string in Tcl is merely one where all the characters it contains are in the range \u0000-\u00FF.

BINARY ENCODE AND DECODE

When encoding binary data as a readable string, the starting binary data is passed to the **binary encode** command, together with the name of the encoding to use and any encoding-specific options desired. Data which has been encoded can be converted back to binary form using **binary decode**. The following formats and options are supported.
base64
The **base64** binary encoding is commonly used in mail messages and XML documents, and uses mostly upper and lower case letters and digits. It has the distinction of being able to be rewrapped arbitrarily without losing information.
During encoding, the following options are supported:

-maxlen *length*
Indicates that the output should be split into lines of no more than *length* characters. By default, lines are not split.

-wrapchar *character*

Indicates that, when lines are split because of the **-maxlen** option, *character* should be used to separate lines. By default, this is a newline character, “\n”.

During decoding, the following options are supported:

-strict
Instructs the decoder to throw an error if it encounters any characters that are not strictly part of the encoding itself. Otherwise it ignores them. RFC 2045 calls for base64 decoders to be non-strict.

hex
The **hex** binary encoding converts each byte to a pair of hexadecimal digits that represent the byte value as a hexadecimal integer. When encoding, lower characters are used. When decoding, upper and lower characters are accepted.
No options are supported during encoding. During decoding, the following options are supported:

-strict
Instructs the decoder to throw an error if it encounters whitespace characters. Otherwise it ignores them.

uuencode
The **uuencode** binary encoding used to be common for transfer of data between Unix systems and on USENET, but is less common these days, having been largely superseded by the **base64** binary encoding.
During encoding, the following options are supported (though changing them may produce files that other implementations of decoders cannot process):

-maxlen *length*
Indicates the maximum number of characters to produce for each encoded line. The valid range is 5 to 85. Line lengths outside that range cannot be accommodated by the encoding format. The default value is 61.

-wrapchar *character*
Indicates the character(s) to use to mark the end of each encoded line. Acceptable values are a sequence of zero or more characters from the set { \x09 (TAB), \x0B (VT), \x0C (FF), \x0D (CR) } followed by zero or one newline \x0A (LF). Any other values are rejected because they would generate encoded text that could not be decoded. The default value is a single newline.

During decoding, the following options are supported:

-strict
Instructs the decoder to throw an error if it encounters anything outside of the standard encoding format. Without this option, the decoder tolerates some deviations, mostly to forgive reflows of lines between the encoder and decoder.

Note that neither the encoder nor the decoder handle the header and footer of the uuencode format.

BINARY FORMAT

The **binary format** command generates a binary string whose layout is specified by the *formatString* and whose contents come from the additional arguments. The resulting binary value is returned.
The *formatString* consists of a sequence of zero or more field specifiers separated by zero or more spaces. Each field specifier is a single type character followed by an optional flag character followed by an optional numeric *count*. Most field specifiers consume one argument to obtain the value to be formatted. The type character specifies how the value is to be formatted. The *count* typically indicates how many items of the specified type are taken from the value. If present, the *count* is a non-negative decimal integer or *, which normally indicates that all of the items in the value are to be used. If the number of arguments does not match the number of fields in the format string that consume arguments, then an error is generated. The flag character is ignored for **binary format**.

Here is a small example to clarify the relation between the field specifiers and the arguments:

```
binary format d3d {1.0 2.0 3.0 4.0} 0.1
```

The first argument is a list of four numbers, but because of the count of 3 for the associated field specifier, only the first three will be used. The second argument is associated with the second field specifier. The resulting binary string contains the four numbers 1.0, 2.0, 3.0 and 0.1.

Each type-count pair moves an imaginary cursor through the binary data, storing bytes at the current position and advancing the cursor to just after the last byte stored. The cursor is initially at position 0 at the beginning of the data. The type may be any one of the following characters:

a
Stores a byte string of length *count* in the output string. Every character is taken as modulo 256 (i.e. the low byte of every character is used, and the high byte discarded) so when storing character strings not wholly expressible using the characters \u0000-\u00ff, the **encoding convertto** command should be used first to change the string into an external representation if this truncation is not desired (i.e. if the characters are not part of the ISO 8859-1 character set.) If *arg* has fewer than *count* bytes, then additional zero bytes are used to pad out the field. If *arg* is longer than the specified length, the extra characters will be ignored. If *count* is *, then all of the bytes in *arg* will be formatted. If *count* is omitted, then one character will be formatted. For example,

```
binary format a7a*a alpha bravo charlie
```

will return a string equivalent to **alpha\000\000bravoc**,

```
binary format a* [encoding convertto utf-8 \u20ac]
```

will return a string equivalent to **\342\202\254** (which is the UTF-8 byte sequence for a Euro-currency character) and

```
binary format a* [encoding convertto iso8859-15 \u20ac]
```

will return a string equivalent to **\244** (which is the ISO 8859-15 byte sequence for a Euro-currency character). Contrast these last two with:

```
binary format a* \u20ac
```

which returns a string equivalent to **\254** (i.e. **\xac**) by truncating the high-bits of the character, and which is probably not what is desired.

A
This form is the same as **a** except that spaces are used for padding instead of nulls. For example,

```
binary format A6A*A alpha bravo charlie
```

will return **alpha bravoc**.

b
Stores a string of *count* binary digits in low-to-high order within each byte in the output string. *Arg* must contain a sequence of **1** and **0** characters. The resulting bytes are emitted in first to last order with the bits being formatted in low-to-high order within each byte. If *arg* has fewer than *count* digits, then zeros will be used for the remaining bits. If *arg* has more than the specified number of digits, the extra digits will be ignored. If *count* is *****, then all of the digits in *arg* will be formatted. If *count* is omitted, then one digit will be formatted. If the number of bits formatted does not end at a byte boundary, the remaining bits of the last byte will be zeros. For example,

```
binary format b5b* 11100 111000011010
```

will return a string equivalent to **\x07\x87\x05**.

B
This form is the same as **b** except that the bits are stored in high-to-low order within each byte. For example,

```
binary format B5B* 11100 111000011010
```

will return a string equivalent to **\xe0\xe1\xa0**.

H
Stores a string of *count* hexadecimal digits in high-to-low within each byte in the output string. *Arg* must contain a sequence of characters in the set "0123456789abcdefABCDEF". The resulting bytes are emitted in first to last order with the hex digits being formatted in high-to-low order within each byte. If *arg* has fewer than

count digits, then zeros will be used for the remaining digits. If *arg* has more than the specified number of digits, the extra digits will be ignored. If *count* is *****, then all of the digits in *arg* will be formatted. If *count* is omitted, then one digit will be formatted. If the number of digits formatted does not end at a byte boundary, the remaining bits of the last byte will be zeros. For example,

```
binary format H3H*H2 ab DEF 987
```

will return a string equivalent to **\xab\x00\xde\xf0\x98**.

h
This form is the same as **H** except that the digits are stored in low-to-high order within each byte. This is seldom required. For example,

```
binary format h3h*h2 AB def 987
```

will return a string equivalent to **\xba\x00\xed\x0f\x89**.

c
Stores one or more 8-bit integer values in the output string. If no *count* is specified, then *arg* must consist of an integer value. If *count* is specified, *arg* must consist of a list containing at least that many integers. The low-order 8 bits of each integer are stored as a one-byte value at the cursor position. If *count* is *****, then all of the integers in the list are formatted. If the number of elements in the list is greater than *count*, then the extra elements are ignored. For example,

```
binary format c3cc* {3 -3 128 1} 260 {2 5}
```

will return a string equivalent to **\x03\xfd\x80\x04\x02\x05**, whereas

```
binary format c {2 5}
```

will generate an error.

s
This form is the same as **c** except that it stores one or more 16-bit integers in little-endian byte order in the output string. The low-order 16-bits of each integer are stored as a two-byte value at the cursor position with the least significant byte stored first. For example,

```
binary format s3 {3 -3 258 1}
```

will return a string equivalent to **\x03\x00\xfd\xff\x02\x01**.

S
This form is the same as **s** except that it stores one or more 16-bit integers in big-endian byte order in the output string. For example,

```
binary format S3 {3 -3 258 1}
```

will return a string equivalent to **\x00\x03\xff\xfd\x01\x02**.

t

This form (mnemonically *tiny*) is the same as **s** and **S** except that it stores the 16-bit integers in the output string in the native byte order of the machine where the Tcl script is running. To determine what the native byte order of the machine is, refer to the **byteOrder** element of the **tcl_platform** array.

i
This form is the same as **c** except that it stores one or more 32-bit integers in little-endian byte order in the output string. The low-order 32-bits of each integer are stored as a four-byte value at the cursor position with the least significant byte stored first. For example,

```
binary format i3 {3 -3 65536 1}
```

will return a string equivalent to **\x03\x00\x00\x00\xfd\xff\xff\xff\x00\x00\x01\x00**

I
This form is the same as **i** except that it stores one or more one or more 32-bit integers in big-endian byte order in the output string. For example,

```
binary format I3 {3 -3 65536 1}
```

will return a string equivalent to **\x00\x00\x00\x03\xff\xff\xff\xfd\x00\x01\x00\x00**

n
This form (mnemonically *number* or *normal*) is the same as **i** and **I** except that it stores the 32-bit integers in the output string in the native byte order of the machine where the Tcl script is running. To determine what the native byte order of the machine is, refer to the **byteOrder** element of the **tcl_platform** array.

w
This form is the same as **c** except that it stores one or more 64-bit integers in little-endian byte order in the output string. The low-order 64-bits of each integer are stored as an eight-byte value at the cursor position with the least significant byte stored first. For example,

```
binary format w 7810179016327718216
```

will return the string **HelloTcl**

W
This form is the same as **w** except that it stores one or more one or more 64-bit integers in big-endian byte order in the output string. For example,

```
binary format Wc 4785469626960341345 110
```

will return the string **BigEndian**

m
This form (mnemonically the mirror of **w**) is the same as **w** and **W** except that it stores the 64-bit integers in the output string in the native byte order of the machine where the Tcl script is running. To determine what the native byte order of the machine is, refer to the **byteOrder** element of the **tcl_platform** array.

f
This form is the same as **c** except that it stores one or more one or more single-precision floating point numbers in the machine's native representation in the output string. This representation is not portable across architectures, so it should not be used to communicate floating point numbers across the network. The size of a floating point number may vary across architectures, so the number of bytes that are generated may vary. If the value overflows the machine's native representation, then the value of FLT_MAX as defined by the system will be used instead. Because Tcl uses double-precision floating point numbers internally, there may be some loss of precision in the conversion to single-precision. For example, on a Windows system running on an Intel Pentium processor,

```
binary format f2 {1.6 3.4}
```

will return a string equivalent to **\xcd\xcc\xcc\x3f\x9a\x99\x59\x40**.

r
This form (mnemonically *real*) is the same as **f** except that it stores the single-precision floating point numbers in little-endian order. This conversion only produces meaningful output when used on machines which use the IEEE floating point representation (very common, but not universal.)

R
This form is the same as **r** except that it stores the single-precision floating point numbers in big-endian order.

d
This form is the same as **f** except that it stores one or more one or more double-precision floating point numbers in the machine's native representation in the output string. For example, on a Windows system running on an Intel Pentium processor,

```
binary format d1 {1.6}
```

will return a string equivalent to **\x9a\x99\x99\x99\x99\x99\xf9\x3f**.

q
This form (mnemonically the mirror of **d**) is the same as **d** except that it stores the double-precision floating point numbers in little-endian order. This conversion only produces meaningful output when used on machines which use the IEEE floating point representation (very common, but not universal.)

Q
This form is the same as **q** except that it stores the double-precision floating point numbers in big-endian order.

x
Stores *count* null bytes in the output string. If *count* is not specified, stores one null byte. If *count* is *, generates an error. This type does not consume an argument. For example,

```
binary format a3xa3x2a3 abc def ghi
```

will return a string equivalent to **abc\000def\000\000ghi**.

X
Moves the cursor back *count* bytes in the output string. If *count* is ***** or is larger than the current cursor position, then the cursor is positioned at location 0 so that the next byte stored will be the first byte in the result string. If *count* is omitted then the cursor is moved back one byte. This type does not consume an argument. For example,

```
binary format a3X*a3X2a3 abc def ghi
```

will return **dghi**.

@
Moves the cursor to the absolute location in the output string specified by *count*. Position 0 refers to the first byte in the output string. If *count* refers to a position beyond the last byte stored so far, then null bytes will be placed in the uninitialized locations and the cursor will be placed at the specified location. If *count* is *****, then the cursor is moved to the current end of the output string. If *count* is omitted, then an error will be generated. This type does not consume an argument. For example,

```
binary format a5@2a1@*a3@10a1 abcde f ghi j
```

will return **abfdeghi\000\000j**.

BINARY SCAN

The **binary scan** command parses fields from a binary string, returning the number of conversions performed. *String* gives the input bytes to be parsed (one byte per character, and characters not representable as a byte have their high bits chopped) and *formatString* indicates how to parse it. Each *varName* gives the name of a variable; when a field is scanned from *string* the result is assigned to the corresponding variable.
As with **binary format**, the *formatString* consists of a sequence of zero or more field specifiers separated by zero or more spaces. Each field specifier is a single type character followed by an optional flag character followed by an optional numeric *count*. Most field specifiers consume one argument to obtain the variable into which the scanned values should be placed. The type character specifies how the binary data is to be interpreted. The *count* typically indicates how many items of the specified type are taken from the data. If present, the *count* is a non-negative decimal integer or *****, which normally indicates that all of the remaining items in the data are to be used. If there are not enough bytes left after the current cursor position to satisfy the current field specifier, then the corresponding variable is left untouched and **binary scan** returns immediately with the number of variables that were set. If there are not enough arguments for all of the fields in the format string that consume arguments, then an error is generated. The flag character "u" may be given to cause some types to be read as unsigned values. The flag is accepted for all field types but is ignored for non-integer fields.

A similar example as with **binary format** should explain the relation between field specifiers and arguments in case of the binary scan subcommand:

```
binary scan $bytes s3s first second
```

This command (provided the binary string in the variable *bytes* is long enough) assigns a list of three integers to the variable *first* and assigns a single value to the variable *second*. If *bytes* contains fewer than 8 bytes (i.e. four 2-byte integers), no assignment to *second* will be made, and if *bytes* contains fewer than 6 bytes (i.e. three 2-byte integers), no assignment to *first* will be made. Hence:

```
puts [binary scan abcdefg s3s first second]
puts $first
puts $second
```

will print (assuming neither variable is set previously):

```
1
25185 25699 26213
can't read "second": no such variable
```

It is *important* to note that the **c**, **s**, and **S** (and **i** and **I** on 64bit systems) will be scanned into long data size values. In doing this, values that have their high bit set (0x80 for chars, 0x8000 for shorts, 0x80000000 for ints), will be sign extended. Thus the following will occur:

```
set signShort [binary format s1 0x8000]
binary scan $signShort s1 val; # val == 0xFFFF8000
```

If you require unsigned values you can include the "u" flag character following the field type. For example, to read an unsigned short value:

```
set signShort [binary format s1 0x8000]
binary scan $signShort su1 val; # val == 0x00008000
```

Each type-count pair moves an imaginary cursor through the binary data, reading bytes from the current position. The cursor is initially at position 0 at the beginning of the data. The type may be any one of the following characters:

a

The data is a byte string of length *count*. If *count* is *****, then all of the remaining bytes in *string* will be scanned into the variable. If *count* is omitted, then one byte will be scanned. All bytes scanned will be interpreted as being characters in the range \u0000-\u00ff so the **encoding convertfrom** command will be needed if the string is not a binary string or a string encoded in ISO 8859-1. For example,

```
binary scan abcde\000fghi a6a10 var1 var2
```

will return **1** with the string equivalent to **abcde\000** stored in *var1* and *var2* left unmodified, and

```
binary scan \342\202\254 a* var1
set var2 [encoding convertfrom utf-8 $var1]
```

will store a Euro-currency character in *var2*.

A
This form is the same as **a**, except trailing blanks and nulls are stripped from the scanned value before it is stored in the variable. For example,

```
binary scan "abc efghi  \000" A* var1
```

will return **1** with **abc efghi** stored in *var1*.

b
The data is turned into a string of *count* binary digits in low-to-high order represented as a sequence of “1” and “0” characters. The data bytes are scanned in first to last order with the bits being taken in low-to-high order within each byte. Any extra bits in the last byte are ignored. If *count* is *****, then all of the remaining bits in *string* will be scanned. If *count* is omitted, then one bit will be scanned. For example,

```
binary scan \x07\x87\x05 b5b* var1 var2
```

will return **2** with **11100** stored in *var1* and **1110000110100000** stored in *var2*.

B
This form is the same as **b**, except the bits are taken in high-to-low order within each byte. For example,

```
binary scan \x70\x87\x05 B5B* var1 var2
```

will return **2** with **01110** stored in *var1* and **1000011100000101** stored in *var2*.

H
The data is turned into a string of *count* hexadecimal digits in high-to-low order represented as a sequence of characters in the set “0123456789abcdef”. The data bytes are scanned in first to last order with the hex digits being taken in high-to-low order within each byte. Any extra bits in the last byte are ignored. If *count* is *****, then all of the remaining hex digits in *string* will be scanned. If *count* is omitted, then one hex digit will be scanned. For example,

```
binary scan \x07\xC6\x05\x1f\x34 H3H* var1 var2
```

will return **2** with **07c** stored in *var1* and **051f34** stored in *var2*.

h
This form is the same as **H**, except the digits are taken in reverse (low-to-high) order within each byte. For example,

```
binary scan \x07\x86\x05\x12\x34 h3h* var1 var2
```

will return **2** with **706** stored in *var1* and **502143** stored in *var2*.
Note that most code that wishes to parse the hexadecimal digits from multiple bytes in order should use the **H** format.

c

The data is turned into *count* 8-bit signed integers and stored in the corresponding variable as a list. If *count* is *****, then all of the remaining bytes in *string* will be scanned. If *count* is omitted, then one 8-bit integer will be scanned. For example,

```
binary scan \x07\x86\x05 c2c* var1 var2
```

will return **2** with **7 -122** stored in *var1* and **5** stored in *var2*. Note that the integers returned are signed, but they can be converted to unsigned 8-bit quantities using an expression like:

```
set num [expr { $num & 0xFF }]
```

s

The data is interpreted as *count* 16-bit signed integers represented in little-endian byte order. The integers are stored in the corresponding variable as a list. If *count* is *****, then all of the remaining bytes in *string* will be scanned. If *count* is omitted, then one 16-bit integer will be scanned. For example,

```
binary scan \x05\x00\x07\x00\xf0\xff s2s* var1 var2
```

will return **2** with **5 7** stored in *var1* and **-16** stored in *var2*. Note that the integers returned are signed, but they can be converted to unsigned 16-bit quantities using an expression like:

```
set num [expr { $num & 0xFFFF }]
```

S

This form is the same as **s** except that the data is interpreted as *count* 16-bit signed integers represented in big-endian byte order. For example,

```
binary scan \x00\x05\x00\x07\xff\xf0 S2S* var1 var2
```

will return **2** with **5 7** stored in *var1* and **-16** stored in *var2*.

t

The data is interpreted as *count* 16-bit signed integers represented in the native byte order of the machine running the Tcl script. It is otherwise identical to **s** and **S**. To determine what the native byte order of the machine is, refer to the **byteOrder** element of the **tcl_platform** array.

i

The data is interpreted as *count* 32-bit signed integers represented in little-endian byte order. The integers are stored in the corresponding variable as a list. If *count* is *****, then all of the remaining bytes in *string* will be scanned. If *count* is omitted, then one 32-bit integer will be scanned. For example,

```
set str \x05\x00\x00\x00\x07\x00\x00\x00\xf0\xff\xff\xff
binary scan $str i2i* var1 var2
```

will return **2** with **5 7** stored in *var1* and **-16** stored in *var2*. Note that the integers returned are signed, but they can be converted to unsigned 32-bit quantities using an expression

like:

```
set num [expr { $num & 0xFFFFFFFF }]
```

I

This form is the same as **I** except that the data is interpreted as *count* 32-bit signed integers represented in big-endian byte order. For example,

```
set str \x00\x00\x00\x05\x00\x00\x00\x07\xff\xff\xff\xf0
binary scan $str I2I* var1 var2
```

will return **2** with **5 7** stored in *var1* and **-16** stored in *var2*.

n

The data is interpreted as *count* 32-bit signed integers represented in the native byte order of the machine running the Tcl script. It is otherwise identical to **i** and **I**. To determine what the native byte order of the machine is, refer to the **byteOrder** element of the **tcl_platform** array.

w

The data is interpreted as *count* 64-bit signed integers represented in little-endian byte order. The integers are stored in the corresponding variable as a list. If *count* is *****, then all of the remaining bytes in *string* will be scanned. If *count* is omitted, then one 64-bit integer will be scanned. For example,

```
set str \x05\x00\x00\x00\x07\x00\x00\x00\xf0\xff\xff\xff
binary scan $str wi* var1 var2
```

will return **2** with **30064771077** stored in *var1* and **-16** stored in *var2*. Note that the integers returned are signed and cannot be represented by Tcl as unsigned values.

W

This form is the same as **w** except that the data is interpreted as *count* 64-bit signed integers represented in big-endian byte order. For example,

```
set str \x00\x00\x00\x05\x00\x00\x00\x07\xff\xff\xff\xf0
binary scan $str WI* var1 var2
```

will return **2** with **21474836487** stored in *var1* and **-16** stored in *var2*.

m

The data is interpreted as *count* 64-bit signed integers represented in the native byte order of the machine running the Tcl script. It is otherwise identical to **w** and **W**. To determine what the native byte order of the machine is, refer to the **byteOrder** element of the **tcl_platform** array.

f

The data is interpreted as *count* single-precision floating point numbers in the machine's native representation. The floating point numbers are stored in the corresponding variable as a list. If *count* is *****, then all of the remaining bytes in *string* will be scanned. If *count* is omitted, then one single-precision floating point number will be scanned. The size of a floating point number may vary across architectures, so the number of bytes that are

scanned may vary. If the data does not represent a valid floating point number, the resulting value is undefined and compiler dependent. For example, on a Windows system running on an Intel Pentium processor,

```
binary scan \x3f\xcc\xcc\xcd f var1
```

will return **1** with **1.6000000238418579** stored in *var1*.

r
This form is the same as **f** except that the data is interpreted as *count* single-precision floating point number in little-endian order. This conversion is not portable to the minority of systems not using IEEE floating point representations.

R
This form is the same as **f** except that the data is interpreted as *count* single-precision floating point number in big-endian order. This conversion is not portable to the minority of systems not using IEEE floating point representations.

d
This form is the same as **f** except that the data is interpreted as *count* double-precision floating point numbers in the machine's native representation. For example, on a Windows system running on an Intel Pentium processor,

```
binary scan \x9a\x99\x99\x99\x99\x99\xf9\x3f d var1
```

will return **1** with **1.6000000000000001** stored in *var1*.

q
This form is the same as **d** except that the data is interpreted as *count* double-precision floating point number in little-endian order. This conversion is not portable to the minority of systems not using IEEE floating point representations.

Q
This form is the same as **d** except that the data is interpreted as *count* double-precision floating point number in big-endian order. This conversion is not portable to the minority of systems not using IEEE floating point representations.

x
Moves the cursor forward *count* bytes in *string*. If *count* is ***** or is larger than the number of bytes after the current cursor position, then the cursor is positioned after the last byte in *string*. If *count* is omitted, then the cursor is moved forward one byte. Note that this type does not consume an argument. For example,

```
binary scan \x01\x02\x03\x04 x2H* var1
```

will return **1** with **0304** stored in *var1*.

X
Moves the cursor back *count* bytes in *string*. If *count* is ***** or is larger than the current cursor position, then the cursor is positioned at location 0 so that the next byte scanned

will be the first byte in *string*. If *count* is omitted then the cursor is moved back one byte. Note that this type does not consume an argument. For example,

```
binary scan \x01\x02\x03\x04 c2XH* var1 var2
```

will return **2** with **1 2** stored in *var1* and **020304** stored in *var2*.

@
Moves the cursor to the absolute location in the data string specified by *count*. Note that position 0 refers to the first byte in *string*. If *count* refers to a position beyond the end of *string*, then the cursor is positioned after the last byte. If *count* is omitted, then an error will be generated. For example,

```
binary scan \x01\x02\x03\x04 c2@1H* var1 var2
```

will return **2** with **1 2** stored in *var1* and **020304** stored in *var2*.

PORTABILITY ISSUES

The **r**, **R**, **q** and **Q** conversions will only work reliably for transferring data between computers which are all using IEEE floating point representations. This is very common, but not universal. To transfer floating-point numbers portably between all architectures, use their textual representation (as produced by **format**) instead.

EXAMPLES

This is a procedure to write a Tcl string to a binary-encoded channel as UTF-8 data preceded by a length word:

```
proc writeString {channel string} {
    set data [encoding convertto utf-8 $string]
    puts -nonewline [binary format Ia* \
            [string length $data] $data]
}
```

This procedure reads a string from a channel that was written by the previously presented *writeString* procedure:

```
proc readString {channel} {
    if {![binary scan [read $channel 4] I length]} {
        error "missing length"
    }
    set data [read $channel $length]
    return [encoding convertfrom utf-8 $data]
}
```

This converts the contents of a file (named in the variable *filename*) to base64 and prints them:

```
set f [open $filename rb]
set data [read $f]
close $f
puts [binary encode base64 -maxlen 64 $data]
```